# C++: Future Directions in Language Innovation


**Herb Sutter**

**TLN309**

**Architect, Developer Division**

**Microsoft Corporation**


PDC05 DEVELOPER POWERED

Microsoft

# Overview

## Key Ongoing Themes

*unmanaged × managed; standards; concurrency*

## A Convergence of Fundamental Building Blocks

*for each (C++/CLI, C++0x, …)*
*automatic type inference (C++0x, Concur, Linq)*
*lambda functions (C++0x?, Concur, Linq)*
*concepts (C++0x, Concur)*

## Concurrency In C++: A Snapshot of Thinking

*Concur basics: active objects; messages; futures*

*loop/data parallel: parallel loops; parallel STL algorithms*

## Summary

# Key Ongoing Themes

**First-class support for both unmanaged and managed:**

- Applying features evenly across the type system (managed features work on unmanaged types, unmanaged features work on managed types) continues to be a principal goal.
- Examples:
  - Further MFC + Windows Presentation Foundation integration.
  - Further unmanaged security work (think /GS on steroids).
  - C++ Linq, both managed and unmanaged. (See Anders' talk!)
  - Improved support for designers, IDE, VSTS (e.g., refactoring).

**Participation in and conformance to open standards:**

- Track new features in ISO C++0x, where x == 9 (we hope).

**Language and library support for concurrency:**

- The Concur project is exploring a set of conforming extensions to C++ to provide higher-level abstractions for concurrent programming.
- Examples: Active objects, messages, futures, parallel loops, parallel STL algorithms.

The rest of this talk

# Overview

## Key Ongoing Themes

*unmanaged × managed; standards; concurrency*

## A Convergence of Fundamental Building Blocks

*for each (C++/CLI, C++0x, …)*
*automatic type inference (C++0x, Concur, Linq)*
*lambda functions (C++0x?, Concur, Linq)*
*concepts (C++0x, Concur)*

## Concurrency In C++: A Snapshot of Thinking

*Concur basics: active objects; messages; futures*

*loop/data parallel: parallel loops; parallel STL algorithms*

## Summary

# For Each

## Why not have loops with greater semantic meaning?

```
for( vector<int>::iterator i = v.begin(); i != v.end(); ++i ) {
  ... do something with *i ...
}
for( int i = 0; i < v.size(); ++i ) {
  ... do something with v[i] ...
}

for each( int vi in v ) {          // C++/CLI syntax
  ... do something with vi ...
}
for( int vi : v ) {   // more likely C++0x syntax
  ... do something with vi ...
}
```

## Commonality:

- Already in C++/CLI, likely to be in ISO C++0x.
- Directly leveraged in Concur. The language can better extend this clearer declaration of intent for new parallelizable loops.

# Automatic Type Inference

**Why tediously tell the compiler what it already knows?**

```
void Print( const map<int,string>& m ) {
  map<int,string>::const_iterator i = m.begin();
  ...
}
```

```
void Print( const map<int,string>& m ) {
  auto i = m.begin();
  ...
}
```

**Commonality:**

- Already voted into ISO C++0x. A significant and broadly applicable convenience, esp. with generic programming.
- Directly leveraged in Concur and Linq. Helps enable an elegant syntax for Concur active lambda functions. The Linq programmer shouldn't be required to, and often can't, write the names of lots of complex anonymous types that get generated on the fly.

# Lambda Functions

## Why write a function object somewhere else?

```
class IsEqualToInt {
  int val;
public:
  IsEqualToInt( int v ) : val(v) { }
  template<typename T> bool operator()( T t ) { return t == val; }
};
find_if( v.begin(), v.end(), IsEqualToInt(3) );    // code lives elsewhere

find_if( v.begin(), v.end(), { return _x == 3; } );     // lambda function
find_if( v.begin(), v.end(), _x == 3 );                 // lambda expression
```

## Commonality:

- Boost.Lambda exists today; language support is a frequently requested extension for ISO C++0x. No serious proposal yet, but together concepts + lambdas take STL to next level.
- Directly leveraged in Concur and Linq. Concur allows lambda function objects to be active, which is a natural abstraction to spin off arbitrary asynchronous work. Linq programmers can easily pass around predicates and other code.

# Concepts

**Why get pages of impenetrable error messages for this?**

```
int i = 0, j = 2;
std::sort( i, j );        // screenfuls of arcane error messages ensue...
```

**Because today sort is defined like this:**

```
template<class RandomAccessIterator>
void sort( RandomAccessIterator first, RandomAccessIterator last );
```

**In C++0x we could say something like:**

```
template<class T> concept RandomAccessIterator {
  ... definition of what a RandomAccessIterator<T> should be able
  to do, such as increment, dereference, etc. ...
};

template<class T>
void sort( RandomAccessIterator<T> first, RandomAccessIterator<T> last );
```

- Enables clear, friendly, and morally decent error messages:

```
int i = 0, j = 2;
std::sort( i, j );        // hey, stupid, i and j aren't RandomAccessIterators...
```

# Concepts (2)

**Concepts will make algorithms more flexible.**

- People always ask why we don't overload algorithms with versions that take a whole container, instead of a [begin,end) range. Answer: Without concepts, the overloads conflict.

```
template<class T>
void sort(      RandomAccessIterator<T> first, RandomAccessIterator<T> last );

template<class T>
void sort(      RandomAccessIterator<T> first, RandomAccessIterator<T> last,
BinaryPredicate<T,T> pred );

template<class T>
void sort(      Container<T> &c );

template<class T>           // without concepts, this one conflicts with the first
void sort(      Container<T> &c, BinaryPredicate<T,T> pred );
```

- To sort a whole container, today we write:

```
sort( v.begin(), v.end() );
```

- With the above, we could write just:

```
sort( v );
```

# Overview

**Key Ongoing Themes**

*unmanaged × managed; standards; concurrency*

**A Convergence of Fundamental Building Blocks**

*for each (C++/CLI, C++0x, …)*
*automatic type inference (C++0x, Concur, Linq)*
*lambda functions (C++0x?, Concur, Linq)*
*concepts (C++0x, Concur)*

**Concurrency In C++: A Snapshot of Thinking**

*Concur basics: active objects; messages; futures*

*loop/data parallel: parallel loops; parallel STL algorithms*

**Summary**

# Concurrency in VS 2005 and Beyond

## Concurrency-related features in Visual Studio 2005:

- OpenMP for loop and data parallel operations.
- Unmanaged and managed memory models for concurrency.
- Cluster debugging.

## The rest of this talk is about futures (pun intended):

- Litmus test: Any prog. model where a Thread is a central abstraction is too low-level (e.g., uncontrolled blocking/reentrancy/affinity).
- Extend today's languages with higher-level abstractions to let developers write correct and efficient concurrent programs in both managed and unmanaged code. This is an "OO for concurrency." Examples: Active objects, futures, parallel std:: algorithms.
- Support the range of concurrency granularities, specifically: a) distributed objects and web services; b) in-box and in-process coarse-grained; and c) fine-grained (loop/data).
- Map abstractions to hardware at run time to defer pooling and scheduling decisions. The key is to reenable the "free lunch" where existing programs get faster on new hardware, by writing apps with lots of latent concurrency, then scaled to actual hardware.

# Illustrating a Principle: Codifying Idioms

## Example: Double-Checked Locking (DCL).

```
volatile Singleton* instance;
Singleton* GetInstance() {
  if( !instance ) {
    // acquire lock
    if( !instance ) {
      instance = new T;
    }
    // release lock
  }
  return instance;
}
```

- Works on some platforms (including VC++ 2005). Read the manual carefully.
- Error-prone. Omit volatile, program compiles & “works.”
- Too low-level. Like coding your own vtables…

## Replacing with a higher-level abstraction:

```
Singleton* instance;
Singleton* GetInstance() {
  once {
      instance = new T;
  }
  return instance;
}
```

## And allowing this variant:

```
Singleton* instance = new T;
Singleton* GetInstance() {
  return instance;
}
```

- Variables should only be initialized once, so we should require the compiler to implicitly Do the Right Thing.

# Overview

## Key Ongoing Themes

*unmanaged × managed; standards; concurrency*

## A Convergence of Fundamental Building Blocks

*for each (C++/CLI, C++0x, …)*
*automatic type inference (C++0x, Concur, Linq)*
*lambda functions (C++0x?, Concur, Linq)*
*concepts (C++0x, Concur)*

## Concurrency In C++: A Snapshot of Thinking

*Concur basics: active objects; messages; futures*

*loop/data parallel: parallel loops; parallel STL algorithms*

## Summary

# 30,000' View: Sources of Concurrency

1. **Active objects with async member function calls.**

```
active C c;
c.f();                  // these calls are nonblocking; each method
c.g();                  // call automatically enqueues a message for c
...                     // this code can execute in parallel with c.f() & c.g()
```

2. **Async calls/work via active lambdas and futures.**
   **Think: "Writing and enqueueing a work item on the fly."**

```
future<int> f1 = active { x.f()+Something() };      // single expression
future<int> f2 = active {
  ...                   // arbitrary
  return ...;           // code
};
int i1 = f1.wait(), i2 = f2.wait();
```

3. **Parallel loops and loop bodies.**

```
active for each( Employee e in emps ) { ... }  // chunk into work items
for( i = v.begin(); i != v.end(); ++i ) active {  // each iter is a work item
  ...
}
```

# Active Objects and Messages

## Nutshell summary:

- active means parallel, wait means blocking:
  Gaining/losing concurrency is explicit, as is reentrancy.
- Each active object conceptually runs on its own thread.
- Method calls from other threads are async messages processed serially ⇒ atomic w.r.t. each other, so no need to lock the object internally or externally. Default mainline is a prioritized FIFO pump.
- Return values and out parameters are futures (future<T>).
- Express thread/task lifetimes as object lifetimes:
  Exploit existing rich language semantics.

```
active class C {
public:
  void f() { … }
};

// in calling code, using a C object
active C c;
c.f();                  // call is nonblocking
…                       // this code can execute in parallel with c.f()
```

# Futures

## Return values are future values:

- Calling an active object method is nonblocking, but a return value (and “out” arguments) cannot be used until an explicit wait for the future to materialize.

```
active class Calc {
  public: double TotalOrders() { return ComputeSomething(); }
};

// in calling code, using a Calc object
future<double> tot = calc.TotalOrders(); // call is nonblocking
... potentially lots of work ...                // parallel work
DoSomethingWith( tot.wait() );                  // explicitly wait to accept
```

## Why require explicit wait? Four compelling reasons:

- No silent loss of concurrency (e.g., early “logFile << tot;”).
- Explicit block point for writing into lent objects (“out” args).
- Explicit point for emitting exceptions.
- Need to be able to pass futures onward to other code (e.g., DoSomethingWith( tot ) ≠ DoSomethingWith( tot.wait() )).

# Example: Multiple Service Requests

**A client sends requests to two different services. Cancel them if a reply is not received from both within a timeout.**

```
active class Service1 {
  public: int Operation1();
};
active class Service2 {
  public: int Operation2();
};
void Process( active Service1& s1, active Service2& s2, int ms ) {
  future<int> result1 = s1.Operation1();
  future<int> result2 = s2.Operation2();
  wait( (result1 && result2) || timeout(ms) );
  if( result1 && result2 ) {
    ...    // use result1 and result2
  else {
    result1.Cancel();   // cancel both
    result2.Cancel();
  }
}
```

# Active Lambda Expressions/Functions

**When the body is an expression or contains a single return statement, the return type is the result of the expression or return statement expression:**

```
x = active { foo(10) };                    // call foo asynchronously
y = active { a->b( c ) };                  // evaluate asynchronously
p = active { new T };    // allocate and construct asynchronously
...  // more code, runs concurrently with all three active lambdas
// there is no waiting until we explicitly say “wait” to use results
return x.wait() * y.wait() * p.wait()->bar();
```

- Writing “active { ... }” is syntactic sugar for writing “active auto ( /* all auto params */ ) { ... }”.

# Using Futures and Active Lambdas

**Active blocks (lambdas) run arbitrary expressions or statement blocks asynchronously.**

- Calling a sync function asynchronously:

```
active { plainObj.Foo(42) }; // type is future<ReturnType>
```

**Idioms:**

- Calling an async function synchronously:

```
activeObj.Bar(3.14).wait();    // type is ReturnType
```

- Calling a function “synchronously” while leaving the calling thread interruptible:

```
active { SomeLongOperation() }.wait();
```

- Creating a callback to do something when the future materializes.

```
future<int> ret = ...;
gcnew active { int i = ret.wait(); DoSomethingWith(i); };
```

# Never Call Unknown Code When…”

**“Never lock when invoking methods on other objects…”**

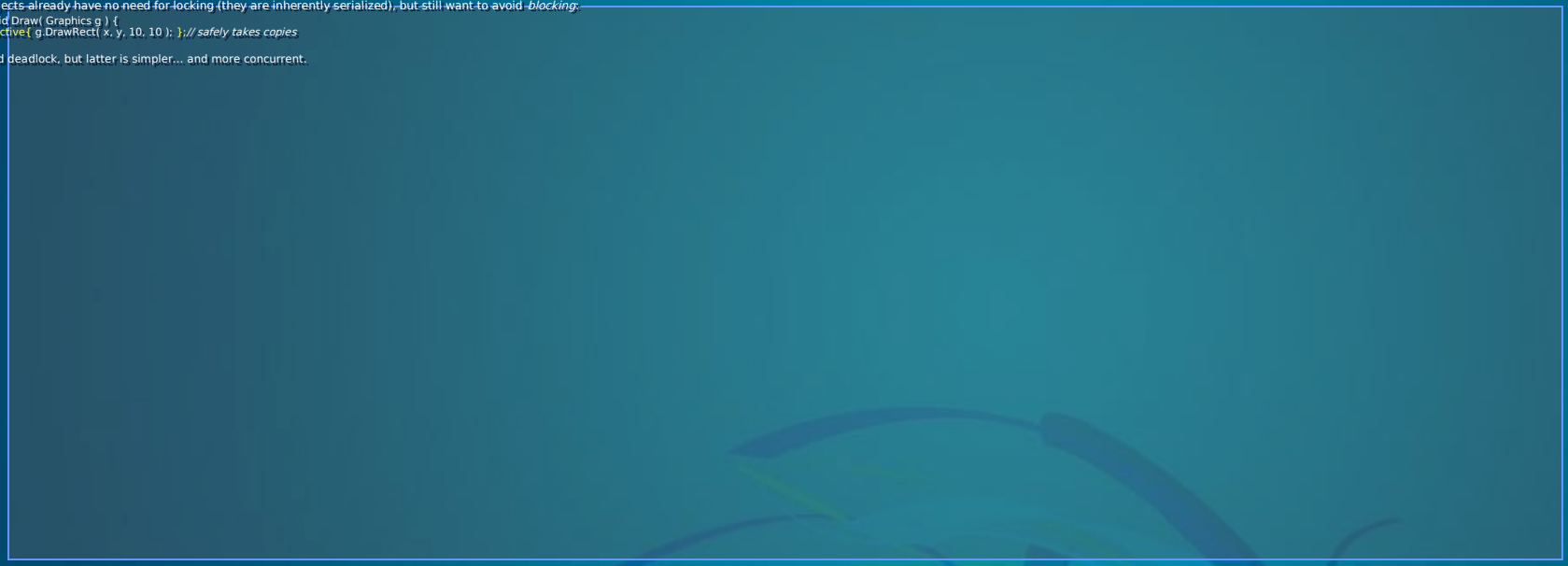
- Example adapted from *CPj2e*:

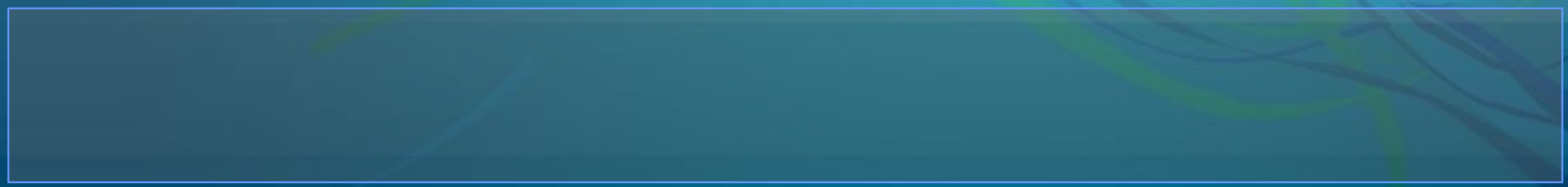
```
class Particle {
  protected int x;                                  // this object's
  protected int y;                                  // coordinates
  ...
  public void Draw( Graphics g ) {
    int localx, localy;
    lock (this) { localx = x; localy = y; }         // safely take copies
    g.DrawRect( localx, localy, 10, 10 );  // call is outside any locks
  }
};
```

- Active objects already have no need for locking (they are inherently serialized), but still want to avoid *blocking*:

```
  public void Draw( Graphics g ) {
    return active{ g.DrawRect( x, y, 10, 10 ); };// safely takes copies
  }
```

- Both avoid deadlock, but latter is simpler… and more concurrent.





# Prevent Blocking Cycles

**Avoid calling unknown code while holding a lock… or from an active object's method.**

- This isn't as drastic as it sounds, because of active lambdas. Just queue up another work item:

```
active ref class Shape {
  int x, y;                                  // this object's coordinates
  ...
public:
  void AddMyselfTo( SyncHashTable^ h ) {
    ... do work ...
    // If we call "h->Add(this,42);" synchronously, it will probably
    // cause a cycle back to "this" (for GetHashCode). Instead:
    return active{ h->Add(this->wait(),42) };// queue up a work item
  }
};
```

# Comparison: FX Async Pattern

## Design Guidelines base async pattern:

- Split single functions into BeginXxx/EndXxx pairs with intermediate structures and explicit callback registrations.
- Sync API:

```
public RetType MethodName(
  Parameters params,
  OutParameters outParams );
```

- Async API:

```
IAsyncResult BeginMethodName(   // caller explicitly calls begin/end
  Parameters params,
  AsyncCallback callback,       // caller must supply callback
  Object state );               // caller usu. must supply cookie
RetType EndMethodName(          // caller explicitly calls begin/end
  IAsyncResult asyncResult,
  OutParameters outParams );
```

# Comparison: FX Async Pattern (2)

**Issues:**

- Intrusive in API: Burying the async work partly inside the APIs changes/bloats the APIs and can duplicate code.
- Complex for callers: Instead of calling one function, the user must call two functions, manage intermediate state, and write an additional function of his own for the callback.
- Hand-coded pattern, no language support/checking: The pattern may vary; consistency depends on manual discipline. This approach has more potential points of failure where the programmer can go wrong.
  - DG example: *"**Do** ensure that the EndMethodName method is always called even if the operation is known to have already completed. This allows exceptions to be received and any resources used in the async operation to be freed."*

# Comparison: Async Pattern + Delegates

```
p

  public static
    int cookieV
    // ... here, c
    int result =
      ((AsyncRe
    Console.Wr
  }
}
```

```
int main() {
  future<int> result = active { sampSyncObj.Square( 42 ) };
  // ... do useful work ...
  Console.WriteLine("Result is {0}", result.wait());
}
```

# Destructor/Dispose Semantics

**Calling an active object's destructor blocks the caller:**

- Till object's execution ends. Only case of blocking w/o "wait."
- Join points are typically at block exits. Supports ganging.

```
{
  active C c;
  …
} // waits for c to complete
```

- Can directly express tasks/process hierarchies (tasks that own other tasks) by associating member lifetimes.

```
active class C {
  active M m;        // embedded member
  …
};
// later
{
  active C c;
  …
} // waits for c & c.m to complete
```

# Example: Producer-Consumer

```
class Consumer {
public:
  void Process( Message msg ) { DoSomethingWith( msg ); }
};
active Consumer consumer;
active class Producer {
  void main() {
    for( int i = 0; ++i < 100; ) {
      Message msg = DoSomeWork();
      consumer.Process( msg );        // non-blocking call
    }
  }
};
int main() {
  active Producer p;
} // wait for p to end
// wait for consumer to end
```

- Impact vs. nonconcurrent: Add `active`.

# Overview

## Key Ongoing Themes

*unmanaged × managed; standards; concurrency*

## A Convergence of Fundamental Building Blocks

*for each (C++/CLI, C++0x, ...)*
*automatic type inference (C++0x, Concur, Linq)*
*lambda functions (C++0x?, Concur, Linq)*
*concepts (C++0x, Concur)*

## Concurrency In C++: A Snapshot of Thinking

*Concur basics: active objects; messages; futures*

*loop/data parallel: parallel loops; parallel STL algorithms*

## Summary

# OpenMP and Beyond

## OpenMP offers many benefits:

- Portable, scalable, flexible, standardized, and performance-oriented interface for parallelizing code.
- Hides many details: Thread team created at app startup, per-thread data allocated when #pragma entered, and work divided into coherent chunks.

## But it also has many limitations:

- C, not C++.
- Outside the language, bolted onto code via #pragmas.
- Only good for simple *for* loops over arrays (doesn't work with STL or BCL collections), and for parallel code sections.
- Doesn't take advantage of C++'s superior abstractions for type safety or generic programming.

# Active Loop Bodies

**Loop body containing only an active lambda is parallel:**

- Loop control statement is evaluated on the master thread.
- Master dynamically assigns loop iterations to team threads.
- Number of threads actually used is tuned to local hardware.

```
for( init; condition; incr ) active {
  // this code is executed across the team
}
for each( Type t in collection ) active {
  // this code is executed across the team
}
while( condition ) active {
  // this code is executed across the team
}
do active {
  // this code is executed across the team
} while( condition );
```

# Active Loops

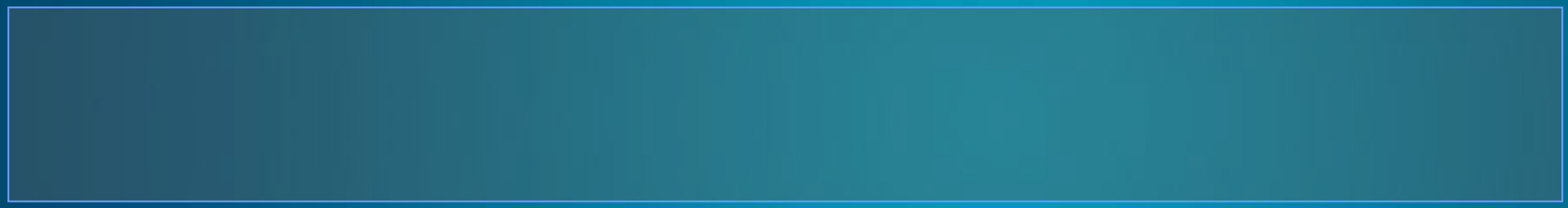
**Entire “for each” loop marked active will divide work:**

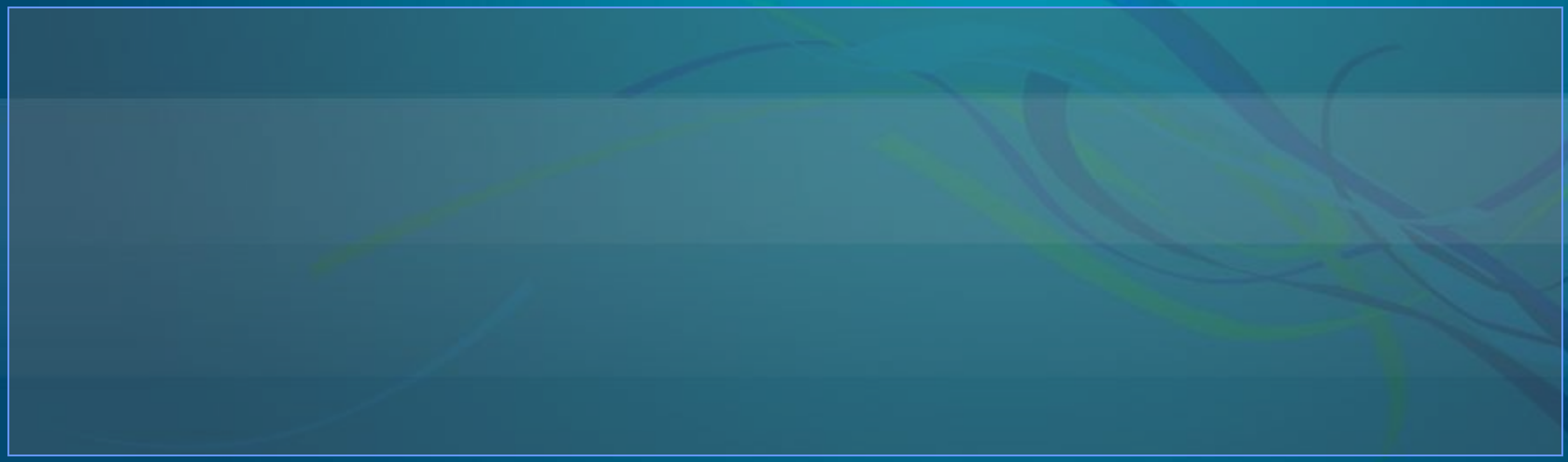
- Loop control statement has to be regular, not customizable.
- Divides work into chunks for master+team threads.
- Number of threads actually used is tuned to local hardware.

```
active for each( Type t in collection ) {
  // each team thread gets a chunk of loop iterations
}
```

- If there are side effects, use futures to merge results:

```
vector<future<...>> results =
  active for each( Type t in collection ) {
  // each team thread gets a chunk of loop iterations
}
for each( future<...> r in results ) {
  // do something with r.wait()
}
```



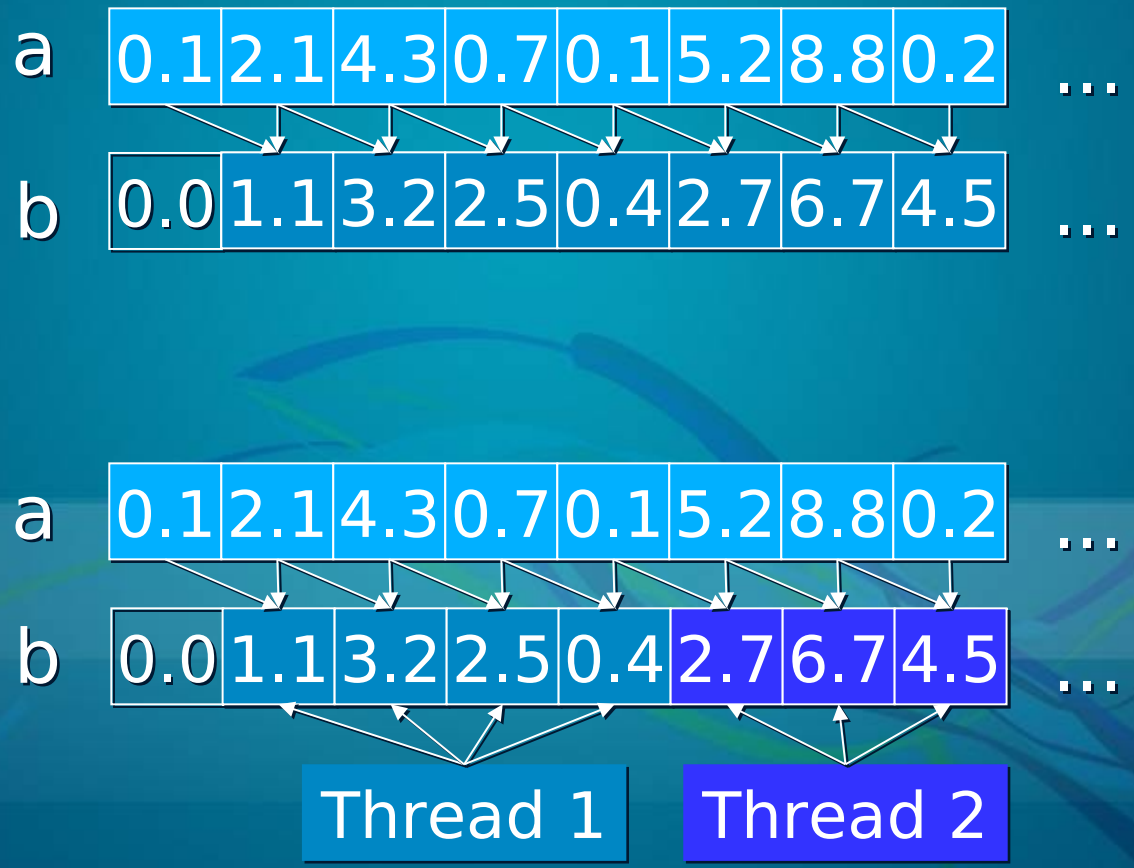


# Example: Average-Neighbors

**Non-concurrent code:**

```
for( i=1; i < n; ++i )
  b[i] = (a[i] + a[i-1]) / 2.0;
```

**What we'd like to be able to do:**



# Example: Average-Neighbors

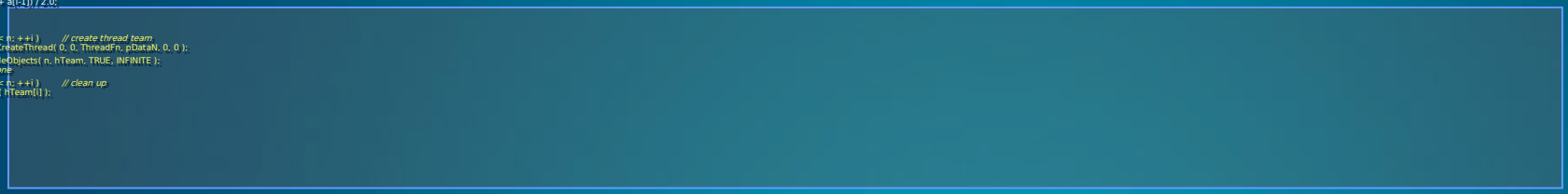
**OpenMP code (example a1):**

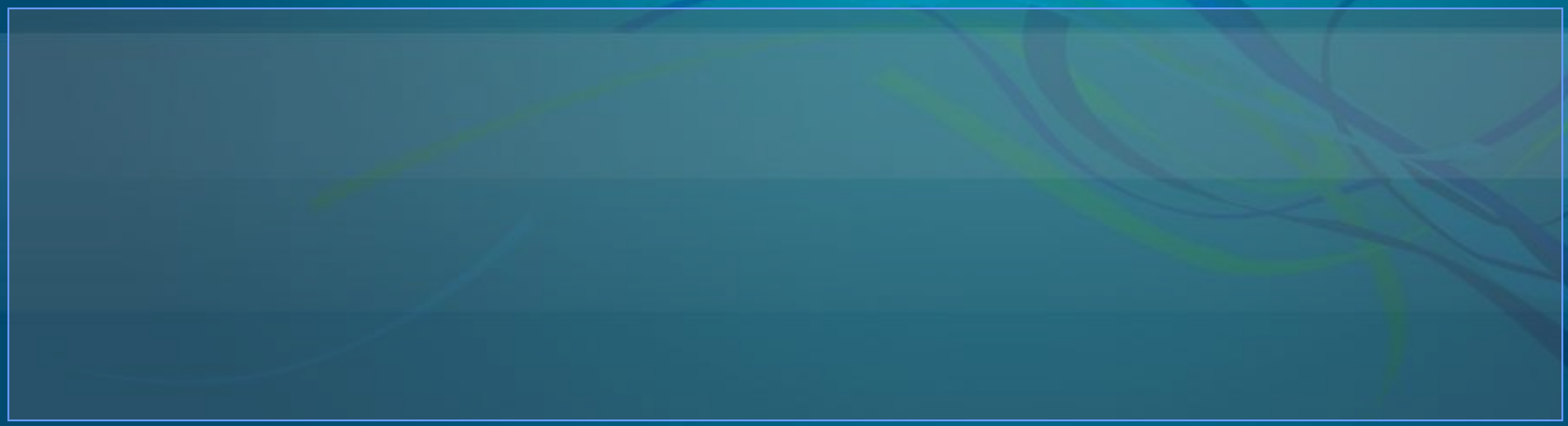
```
#pragma omp parallel for
for( i=1; i < n; ++i )
  b[i] = (a[i] + a[i-1]) / 2.0;
```

**Explicit Windows threads code:**

```
DWORD ThreadFn( VOID* pData ) { // primary function
  for( int i = pData->Start; i < pData->Stop; ++i )
    b[i] = (a[i] + a[i-1]) / 2.0;
  return 0;
}

for( int i=0; i < n; ++i )        // create thread team
  hTeam[i] = CreateThread( 0, 0, ThreadFn, pDataN, 0, 0 );
WaitForMultipleObjects( n, hTeam, TRUE, INFINITE );
// wait until done
for( int i=0; i < n; ++i )        // clean up
  CloseHandle( hTeam[i] );
```





# Example: Average-Neighbors

**OpenMP code (example a1):**

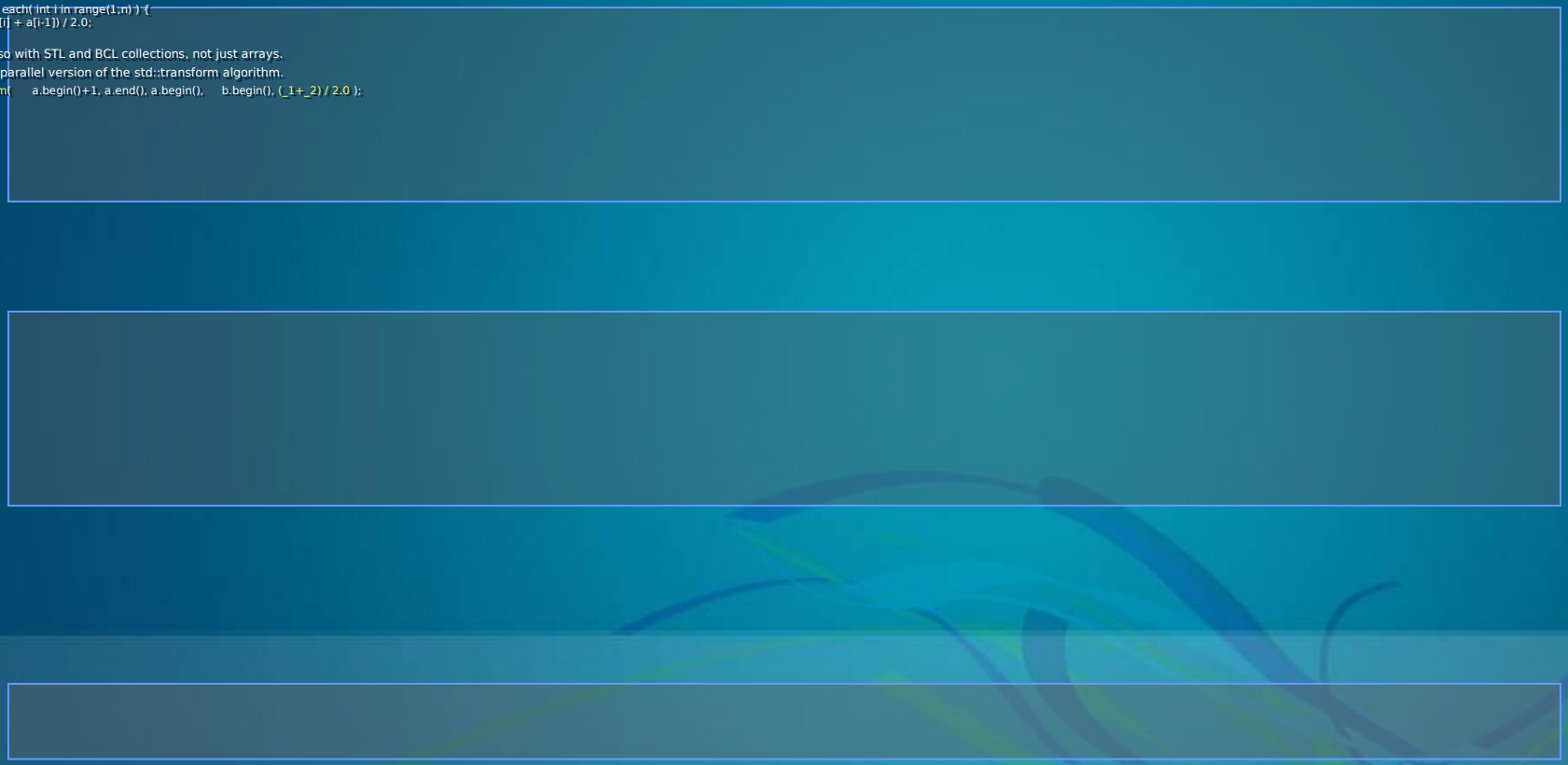
```
#pragma omp parallel for
for( i=1; i < n; ++i )
  b[i] = (a[i] + a[i-1]) / 2.0;
```

**Concur code:**

```
active for each( int i in range(1,n) ) {
  b[i] = (a[i] + a[i-1]) / 2.0;
}
```

- Works also with STL and BCL collections, not just arrays.
- Or use a parallel version of the std::transform algorithm.

```
ptransform(     a.begin()+1, a.end(), a.begin(),     b.begin(), (_1+_2) / 2.0 );
```



# Example: Parallel Array Update

**OpenMP code:**

```
void a7( float *x, int *y, int n ) {
  float a = 0.0;
  int b = 0, i;
  #pragma omp parallel for reduction(+:a) reduction(^:b)
  for( i=0; i<n; i++ ) {
    a += x[i];
    b ^= y[i];
  }
}
```

**Concur code:**

```
void a7( vector<float> &x, vector<int> &y ) {
  float       a = accumulate( x, 0.0,  _1 + _2 );
  int         b = accumulate( y, 0,    _1 ^ _2 );
}
```

- Also, may get better cache performance by going through one collection and then the other. (The original code may have bad cache behavior unless x and y are related in some way.)

# Example: Loop Parallelization

**OpenMP code:**

```
void a12( float *x, int *index, int n ) {
  int i;
  #pragma omp parallel for shared(x, y, index, n)
  for( i = 0; i < n; i++ ) {
    #pragma omp atomic
    x[ index[i] ] += work1( i );
  }
}
```

**Concur code:**

```
void a12( vector<float> &x, const vector<int> &index ) {
  active for each( int i in range(0, index.size()) ) {
    int ii = index[i];
    float f = work1( i );
    atomic { x[ ii ] += f; }      // compiler warning if atomic is absent
  }
}
```

# Overview

## Key Ongoing Themes

*unmanaged × managed; standards; concurrency*

## A Convergence of Fundamental Building Blocks

*for each (C++/CLI, C++0x, …)*
*automatic type inference (C++0x, Concur, Linq)*
*lambda functions (C++0x?, Concur, Linq)*
*concepts (C++0x, Concur)*

## Concurrency In C++: A Snapshot of Thinking

*Concur basics: active objects; messages; futures*

*loop/data parallel: parallel loops; parallel STL algorithms*

## Summary

# Key Ongoing Themes

**First-class support for both unmanaged and managed:**

- Applying features evenly across the type system (managed features work on unmanaged types, unmanaged features work on managed types) continues to be a principal goal.
- Examples:
  - Further MFC + Windows Presentation Foundation integration.
  - Further unmanaged security work (think /GS on steroids).
  - C++ Linq, both managed and unmanaged. (See Anders' talk!)
  - Improved support for designers, IDE, VSTS (e.g., refactoring).

**Participation in and conformance to open standards:**

- Track new features in ISO C++0x, where x == 9 (we hope).

**Language and library support for concurrency:**

- The Concur project is exploring a set of conforming extensions to C++ to provide higher-level abstractions for concurrent programming.
- Examples: Active objects, messages, futures, parallel loops, parallel STL algorithms.

The bulk of this talk

# Concurrency in VS 2005 and Beyond

## Concurrency-related features in Visual Studio 2005:

- OpenMP for loop and data parallel operations.
- Unmanaged and managed memory models for concurrency.
- Cluster debugging.

## The bulk of this talk was about futures (pun intended):

- Litmus test: Any programming model where a Thread (incl. thread pool) is a central abstraction is too low-level.
- Extend today's languages with higher-level abstractions to let developers write correct and efficient concurrent programs in both managed and unmanaged code. This is an "OO for concurrency."
  Examples: Active objects, futures, parallel std:: algorithms.
- Support the range of concurrency granularities, specifically:
  a) distributed objects and web services; b) in-box and in-process coarse-grained; and c) fine-grained (loop/data).
- Map abstractions to hardware at run time to defer pooling and scheduling decisions. The key is to reenable the "free lunch" where existing programs get faster on new hardware, by writing apps with lots of latent concurrency, then scaled to actual hardware.

# Community Resources

**If you're reading this deck early, at PDC go see:**

- FUN302 - Programming with Concurrency, Part 1 (Tue 2:45pm)
- FUN405 - Programming with Concurrency, Part 2 (Tue 4:15pm)
- TLN307 - C#: Future Directions in Language Innovation from Anders Hejlsberg (Wed 3:15pm)
- PRS313 - Windows Presentation Foundation ("Avalon"): Integrating with Your Win32/MFC Application (Wed 5:00pm)
- FUN 323 - Microsoft Research: Future Possibilities in Concurrency (Fri 8:30am)

**After PDC:**

- If you missed those sessions, watch them on the DVD.
- MSDN dev center: http://msdn.microsoft.com/webservices/
- MSDN  Visual C++ Forums: http://forums.microsoft.com/msdn/default.aspx?ForumGroupID=8
- Channel 9 tag: http://channel9.msdn.com/tags/C++

# Questions?